Java assignment – 10

```
import java.util.*;

public class SortUtil {
    public static <T extends Comparable<T>> void sortList(List<T> list) {
        Collections.sort(list);
    }

    public static void main(String[] args) {
        List<String> stringList = Arrays.asList("banana", "apple", "cherry");
        List<Integer> intList = Arrays.asList(3, 1, 2);

        sortList(stringList);
        sortList(intList);

        System.out.println("Sorted String List: " + stringList);
        System.out.println("Sorted Integer List: " + intList);
    }
}
```

Output:

```
Sorted String List: [apple, banana, cherry]
Sorted Integer List: [1, 2, 3]
```

```
import java.util.ArrayList;
import java.util.List;

class TreeNode<T> {
    T value;
    List<TreeNode<T>> children;

    public TreeNode(T value) {
        this.value = value;
        this.children = new ArrayList<>();
    }

    public void addChild(TreeNode<T> child) {
        children.add(child);
    }

    public void dfs(TreeNode<T> node) {
        System.out.println(node.value);
        for (TreeNode<T> child : node.children) {
            dfs(child);
        }
    }

    public TreeNode<T> find(TreeNode<T> node, T value) {
        if (node.value.equals(value)) {
            return node;
        }
        for (TreeNode<T> child : node.children) {
            TreeNode<T> result = find(child, value);
```

```
            if (result != null) {
                return result;
            }
        }
        return null;
    }

    public static void main(String[] args) {
        TreeNode<String> root = new TreeNode<>("root");
        TreeNode<String> child1 = new TreeNode<>("child1");
        TreeNode<String> child2 = new TreeNode<>("child2");
        root.addChild(child1);
        root.addChild(child2);
        child1.addChild(new TreeNode<>("child1.1"));
        child1.addChild(new TreeNode<>("child1.2"));

        System.out.println("DFS Traversal:");
        root.dfs(root);

        System.out.println("Find Node:");
        TreeNode<String> foundNode = root.find(root, "child1.1");
        System.out.println(foundNode != null ? foundNode.value : "Node not found");
    }
}
```

Output:

```
DFS Traversal:
root
child1
child1.1
child1.2
child2
Find Node:
child1.1
```

```
import java.util.PriorityQueue;

class GenericPriorityQueue<T extends Comparable<T>> {
    private PriorityQueue<T> queue;

    public GenericPriorityQueue() {
        queue = new PriorityQueue<>();
    }

    public void enqueue(T element) {
        queue.add(element);
    }

    public T dequeue() {
        return queue.poll();
    }

    public T peek() {
        return queue.peek();
```

```
    }

    public static void main(String[] args) {
        GenericPriorityQueue<Integer> intQueue = new GenericPriorityQueue<>();
        intQueue.enqueue(5);
        intQueue.enqueue(1);
        intQueue.enqueue(3);
        System.out.println("Dequeue: " + intQueue.dequeue());
        System.out.println("Peek: " + intQueue.peek());

        GenericPriorityQueue<String> stringQueue = new GenericPriorityQueue<>();
        stringQueue.enqueue("apple");
        stringQueue.enqueue("banana");
        stringQueue.enqueue("cherry");
        System.out.println("Dequeue: " + stringQueue.dequeue());
        System.out.println("Peek: " + stringQueue.peek());
    }
}
```

Output:

```
Dequeue: 1
Peek: 3
Dequeue: apple
Peek: banana
```

```
import java.util.*;

class Graph<T> {
    private Map<T, List<T>> adjList;
```

```
private boolean isDirected;

public Graph(boolean isDirected) {
    this.adjList = new HashMap<>();
    this.isDirected = isDirected;
}

public void addNode(T node) {
    adjList.putIfAbsent(node, new ArrayList<>());
}

public void addEdge(T src, T dest) {
    adjList.get(src).add(dest);
    if (!isDirected) {
        adjList.get(dest).add(src);
    }
}

public void bfs(T start) {
    Set<T> visited = new HashSet<>();
    Queue<T> queue = new LinkedList<>();
    queue.add(start);
    visited.add(start);

    while (!queue.isEmpty()) {
        T node = queue.poll();
        System.out.println(node);
        for (T neighbor : adjList.get(node)) {
            if (!visited.contains(neighbor)) {
```

```
                queue.add(neighbor);
                visited.add(neighbor);
            }
        }
    }
}

public void dfs(T start) {
    Set<T> visited = new HashSet<>();
    Stack<T> stack = new Stack<>();
    stack.push(start);

    while (!stack.isEmpty()) {
        T node = stack.pop();
        if (!visited.contains(node)) {
            System.out.println(node);
            visited.add(node);
            for (T neighbor : adjList.get(node)) {
                stack.push(neighbor);
            }
        }
    }
}

public static void main(String[] args) {
    Graph<String> graph = new Graph<>(false);
    graph.addNode("A");
    graph.addNode("B");
    graph.addNode("C");
```

```
        graph.addEdge("A", "B");
        graph.addEdge("A", "C");
        graph.addEdge("B", "C");

        System.out.println("BFS Traversal:");
        graph.bfs("A");

        System.out.println("DFS Traversal:");
        graph.dfs("A");
    }
}
```

Output:

```
BFS Traversal:
A
B
C
DFS Traversal:
A
C
B
```

```
class Matrix<T extends Number> {
    private T[][] data;

    public Matrix(T[][] data) {
        this.data = data;
    }

    public Matrix<T> add(Matrix<T> other) {
```

```
    int rows = data.length;
    int cols = data[0].length;
    T[][] result = (T[][]) new Number[rows][cols];

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            result[i][j] = (T) Double.valueOf(data[i][j].doubleValue() + other.data[i][j].doubleValue());
        }
    }
    return new Matrix<>(result);
}

public Matrix<T> subtract(Matrix<T> other) {
    int rows = data.length;
    int cols = data[0].length;
    T[][] result = (T[][]) new Number[rows][cols];

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            result[i][j] = (T) Double.valueOf(data[i][j].doubleValue() - other.data[i][j].doubleValue());
        }
    }
    return new Matrix<>(result);
}

public Matrix<T> multiply(Matrix<T> other) {
    int rows = data.length;
    int cols = other.data[0].length;
    T[][] result = (T[][]) new Number[rows][cols];
```

```
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                result[i][j] = (T) Double.valueOf(0);
                for (int k = 0; k < data[0].length; k++) {
                    result[i][j] = (T) Double.valueOf(result[i][j].doubleValue() + data[i][k].doubleValue() *
other.data[k][j].doubleValue());
                }
            }
        }
        return new Matrix<>(result);
    }

    public void print() {
        for (T[] row : data) {
            for (T val : row) {
                System.out.print(val + " ");
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        Integer[][] intData1 = {{1, 2}, {3, 4}};
        Integer[][] intData2 = {{5, 6}, {7, 8}};
        Matrix<Integer> intMatrix1 = new Matrix<>(intData1);
        Matrix<Integer> intMatrix2 = new Matrix<>(intData2);

        System.out.println("Integer Matrix Addition:");
```

```
        Matrix<Integer> intResult = intMatrix1.add(intMatrix2);
        intResult.print();

        Double[][] doubleData1 = {{1.1, 2.2}, {3.3, 4.4}};
        Double[][] doubleData2 = {{5.5, 6.6}, {7.7, 8.8}};
        Matrix<Double> doubleMatrix1 = new Matrix<>(doubleData1);
        Matrix<Double> doubleMatrix2 = new Matrix<>(doubleData2);

        System.out.println("Double Matrix Multiplication:");
        Matrix<Double> doubleResult = doubleMatrix1.multiply(doubleMatrix2);
        doubleResult.print();
    }
}
```

Output:

```
Integer Matrix Addition:
6.0 8.0
10.0 12.0
Double Matrix Multiplication:
22.990000000000002 26.620000000000005
52.03 60.5
```